# THE ROLE OF OVERLAY SERVICES IN A SELF-MANAGING FRAMEWORK FOR DYNAMIC VIRTUAL ORGANIZATIONS

Per Brand, Joel Höglund and Konstantin Popov
*SICS, Sweden*
{perbrand,joel,kost}@sics.se

Noel de Palma, Fabienne Boyer and Nikos Parlavantzas
*INRIA, France*
{noel.depalma,fabienne.boyer,nikolaos.parlavantzas}@inrialpes.fr

Vladimir Vlassov and Ahmad Al-Shishtawy
*KTH, Sweden*
{vladv,ahmadas}@kth.se

**Abstract** We combine and extend recent results in autonomic computing and structured peer-to-peer to build an infrastructure for constructing and managing dynamic virtual organizations. The paper focuses on the middle layer of the proposed infrastructure, in-between the Niche overlay system on the bottom, and an architecture-based management system based on Jade on the top. The middle layer, the overlay services, are responsible for all sensing and actuation carried out by the VO management. We describe in detail the API of the resource and component overlay services both on the management node and the nodes hosting resources. We present a simple use case demonstrating resource discovery, initial deployment, self-configuration as a result of resource availability change, self-healing, self-tuning and self-protection. The advantages of the design are 1) the overlay services are in themselves self-managing, and sensor/actuation services they provide are robust, 2) management can be dealt with declaratively and at a high-level, and 3) the overlay services provide good scalability in dynamic VOs.

## 1. Introduction

The context of this work is the effort to combine, integrate and extend recent results in autonomic computing with structured peer-to-peer systems. The ultimate goal is to build an infrastructure for constructing and managing dy-

namic collaborative virtual organizations (VOs) for resource sharing. This paper focuses on the middle layer of this infrastructure, a number of vital VO-management services. We outline the design of these *overlay services*, and describe in detail two of them. We also briefly show how these services interface with high-level management functions and the underlying structured peer-to-peer system.

The dynamic VOs that we target are Internet-based. The dynamism is along two dimensions, *churn* and *evolution*. Churn means that the identities of the individual members of the VO and the resources that they bring into the VO are constantly changing. Under churn only, the totality of resources and members remains the same. Evolution means that the number or amount of resources and members also changes. There are also aspects of both churn and evolution regarding the types and usage of services that the VO is running and VO policies.

Self-management (or autonomic computing) is actively pursued as human system administration is expensive, error-prone, and often non-optimal. There is a considerable body of work in this area, and some progress has been made. In dynamic VOs with high rates of churn and evolution self-management becomes crucial. The overlay services enable self-management of the VO.

The infrastructure for managing dynamic VOs can be split into three layers. The *topmost* layer, the management policy/logic layer, consists of high-level management functions and tools. This let VO managers set appropriate policies for applications and services that are run in the VO. This includes aspects of application configuration, healing, and tuning, as well as policies that prioritize between applications/services upon resource contention. The *bottom-most* layer is a self-organizing overlay network called Niche that connects all machines/resources in the VO. Niche is based on a DHT, and includes a publish/subscribe service. The *middle* layer, the overlay services are the focus of this paper. They provide VO management services such as discovery, resource monitoring, member monitoring, and deployment of components.

The paper is organized as follows. First we describe VO management. Then we give a high-level description of the architecture. Thereafter, in a bottom-up fashion, we describe the three layers, beginning with Niche, followed by the overlay services and finally the management policy/logic level. Only the overlay services are described in detail. We then present a simple use case, involving deploying an application and instrumenting appropriate self-* policies. The use case concentrates on the interaction between the overlay services and the management logic. Finally, we conclude and relate to other work.

## 2. VO Management

Within the framework set by VO policy, members provide resources and services to the VO. VO management monitors, aggregates, presents and controls

these resources and services to/for the VO members. Services are also created within the VO making use of the aggregated resources. The VO management is thus responsible for deploying and managing applications that make use of aggregated computation and/or storage facilities. Managing these applications, presented to members as services, in the face of churn will require frequent management interventions. We can divide the concerns of VO management into management of four different kinds of entities.

***Resources*** here are defined as that which members bring into the VO. It represents such basic things as computation power and data storage. The VO must be able to aggregate resources upon need.

***Components*** here are defined as the constituent parts of services that are created in the VO by utilization of aggregated resources. The VO may deploy a distributed application consisting of several components on a number of member machines and then maintains the application in the face of individual machine failure or disconnection. 'Components' are used here in a generic sense, and could include, for instance, storage 'components' used in constructing a data repository service.

***Members*** come and go. Policy dictates the type of member roles that exist within the VO. With a given role there are both obligations and privileges, and it is part of monitoring to check that members meet their obligations.

***Services***: The VO management ensures that the services provided by the VO are made available in such a way to allow members to discover and use them.

We illustrate the four kinds of entities as follows: member M joins a VO and provides the resource R during certain hours. The VO makes use of the resource R and deploys some component C on R as part of an application A. It also publishes and wraps A so that other members can discover and use A.

# 3. Architecture

## 3.1 Management Logic

Our approach to self-management for dynamic VOs is the architecture-based control one and based on earlier work on the Fractal component model and the Jade management system [4]. Our system has one or more manager components that are continuously monitoring (through sensors) and controlling (via actuators) the VO in a feedback loop in accordance with high-level policies/goals and system administration input. Self-management includes self-healing, self-tuning, self-configuration, and self-protection.

The control approach of management distinguishing between three aspects of management:

**Sensing**: the ability to sense or observe the state of system and system elements. In general observation may be active (triggered by the observer) or passive (triggered by the element).

**Actuation**: the ability to control and affect the system elements.
**Decision**: the logic that given knowledge of the system elements (provided by sensing) decides on actions (done by actuation) to ensure proper operation of the system. This ranges from simple rules to sophisticated AI techniques.

An architecture description language (ADL) is used to specify declaratively the initial deployment and simple self-configuration and self-healing behaviours of the system. Architectures are specified in terms of components and bindings. Component descriptions include requirements and preferences for resources necessary for deployment of the component. Component descriptions state also component properties crucial for management logic, such as whether the component's state can be extracted into a data structure.

Other self-* behaviours are specified in terms of events and handlers and abstractions thereof. Events reflect status changes in the VO, such as availability of resources, and status changes of application components, such as failures. Event handlers evaluate the status of the application and the environment and can replace, add and remove application components and bindings.

The high-level ADL descriptions are compiled into the low-level management logic assembler that utilizes the VO-management overlay services. While the ADL descriptions refer to architecture-level notions, the assembler code works with mutable references to low-level entities such as resource and component handlers, entity and VO status watchers, and stateful event handlers.

## 3.2 Overlay Services

The overlay services are primarily sensor and actuation services (SA in Figure 1). They also provide the infrastructure for delegation of management logic. All nodes in the system are 'known' to the SA services and are part of the same overlay. When nodes enter or leave the system they join/leave the overlay. When resources join/leave the system they report this to their local overlay proxy, an action that may lead to a management action as some management rule is triggered. Resources are also monitored for failures. An actuating command such as deploying a component on a given resource is issued on a management node and will eventually reach a correct managed node, where the deployment is trigged using the managed node API.

The figure on the left shows the system architecture at management nodes. The management logic senses and actuates through the three overlay services through well-specified management APIs.

1. Resource sensor and actuator service
2. Component/service sensor and actuator service
3. Member sensor and actuator service

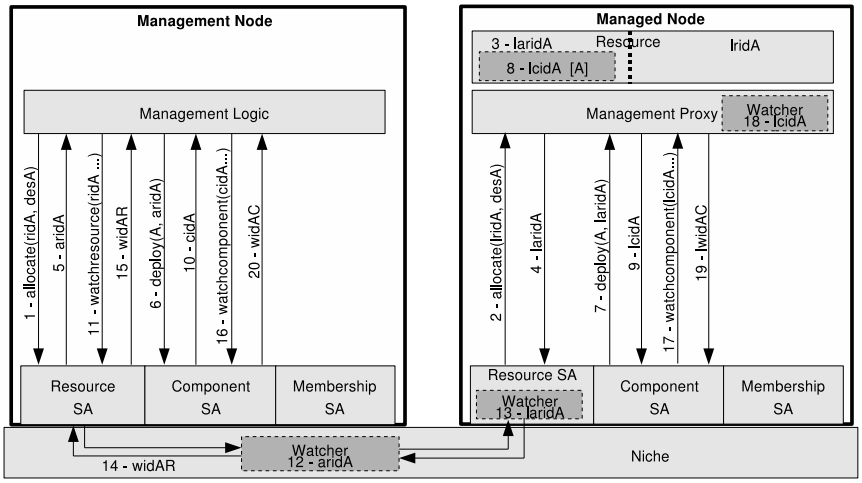


*Figure 1.* System Architecture and the Use Case.

The three services are reflected in three front-end subsystems that perform only a small amount of computation, packaging and bookkeeping. Communication with other nodes takes place exclusively in the Niche layer.

The figure on the right shows the system architecture at managed nodes. The Management Proxy component interacts with the three overlay services through the services' managed node API interfaces. The proxy interacts with resources and components allocated/located on the local machine on behalf of the VO. The interface between the management proxy and the overlay services is used in particular for 1) notifications of resources joining and leaving the VO as the owning members withdraws and add resources to the VO and 2) components communicating via bindings with other components currently residing on remote nodes. A physical node may be both a manager and a managed one.

## 4. Niche

Overlay services described in this paper exploit the Distributed K-ary System (DKS) [1, 3]middleware and its extension called Niche. DKS has a circular identifier space, similarly to Chord [17]. Each node is responsible for IDs in the interval between its own ID and the ID of its predecessor. A message sent to a DKS ID is received by the node responsible for that ID.

DKS self-organizes itself as nodes join, leave and fail. The join and leave operations are locally atomic. Symmetric replication [10]distributes replicas symmetrically among DKS nodes and enables concurrent requests improving efficiency and fault-tolerance. DKS provides broadcast and multicast [2, 8].

Niche extends the DKS and provides in particular the "set of network references" abstraction, SNR hereafter. A SNR keeps a set of references to abstract entities. Individual references in a SNR can be accessed by their SNR-specific IDs. References to entities can be updated. SNR assumes that clients that receive out-of-date references as a result of a concurrent read operation can recognize the problem. For example, if the entity represents a component that is relocated to another node, an attempt to access the previous location of the component will result with an "out-of-date reference" error. Concurrent reads and updates can be also controlled with a conditional update operation that proceeds only if the reference supplied as an additional parameter is equal to the reference currently held in the SNR.

A SNR also monitors the status of its entities and can notify clients that are subscribed to entity status updates. SNR entities can communicate their status to SNR, either by "status polling" by the SNR or by asynchronous messages sent to SNR. SNR failure monitoring of resources is polling-based, using existing DKS functionality. Resource status updates such as resource leaves is communicated to the SNR using messages.

SNRs are reliable and scalable using replication. Sets of references are identified by Niche IDs. Each SNR set element contains the reference ID and the reference itself. A SNR contains also a possibly empty set of references to Niche entities that are subscribed for reference status updates. This set of subscriptions is shared by all references in the SNR. SNRs are stored on Niche nodes responsible for their set IDs. SNRs are replicated on a number of consecutive Niche nodes starting from the primary replica. When a primary replica for a SNR ID fails or leaves, the next replica automatically becomes Niche-responsible for the ID and thus becomes the new primary replica.

## 5. Overlay Services

The elements of the management logic assembler can be divided into categories based on their place in the management feedback loop.

**Events**: Sensing is realized through events.

**Sensor installation**: This instruments the sensing, which can be seen as publish directives. Discovery operations and watchers belong here, where management asks the overlay services to find resources (active sensing) or monitor specific entities (passive sensing), respectively. Information on entities will only be available if there are watchers installed.

**Triggers**: This is an actuating part of the management logic whereby a command is sent to a specific entity or groups of entities.

**Activation of event handlers**: Activating an event handler may be seen as a subscription. Event handlers can be created and/or stopped. For one event there may be more than one event handler, which may be triggered in arbitrary order.

## 5.1 Management API

In the following section the management API is described in an abstract form. We describe the more important functions of the resource and component overlay services and all operations used in the use case section.

We use VO-wide identifiers for resources, components, watchers, bindings and groups. Futures are used to simplify data-flow dependencies. When created, a future represents an unknown value which will be instantiated, which allows waiting for that value. For most futures a failure indication is a possible value. Futures are identified by a capital F in the variable name.

**Sensor Installation**

```
wid:discoverResource(req, compare, tCompare, class, curRes)
widF:watchComponent(cid, compParam, compare, tCompare)
widF:watchResource(rid, resParam, compare, tCompare)
stopWatcher(wid)
```

These instructions install sensors which will continuously report about resources matching given requirements or component and resource changes specified by parameters. For a sensor to report a change, the change has to be significant, as calculated by the given `compare`-function, and a threshold function `tCompare` that determines if a resource is sufficiently better than `curRes`.

**Triggers**

```
aridF:allocate(rid, specification)
boolF:deallocate(arid)
cidF:deploy(component, arid)
fcidF:passivate(cidA)
fcidF:checkpoint(cidA)
boolF:start(cid)
gcidF:group(listOfComponentIds)
boolF:addToGroup(gid, cid)
bidF:bind(cid, gcid, bindDescSource, bindDescDestination, type)
boolF:unbind(bid)
rid:oneShotResourceDiscover(req, compare)
```

The deploy trigger is overloaded. The argument can be code, a checkpoint, url, etc. The data associated with passivation is stored in Niche under `fcidF`. Bindings are assumed to be unidirectional and asynchronous. If needed, binding descriptions for the sending and delivering side give additional information to connect the components.

**Events**

```
resourceReport(wid, oldState, newState)
componentReport(wid, componentParam, oldValue, newValue)
discoveryReport(wid, rid, resourceDescription)
```

Resource and component report correspond to watch subscriptions, while discovery report corresponds to discoverResource subscriptions. They are generated if the change has triggered the initially given threshold function.

**Event Handlers**

```
upon event eventName(wid, es) with <attributes> do
activateEventHandler(rule, wid, initAttributes)
passivateEventHandler(rule, wid)
```

The event handler is triggered by an exact match on both the event name, and the value of the id, wid. Es represents parameters given in the event, attributes represents parameters given when instantiating the event handler.

## 5.2 Managed Node Side API

The managed node side of the API works with local ids. The sensor and actuator services will do the conversion between local and global ids.

**Downcalls, Initiated by Management Proxy Layer**

```
resourceJoin(lrid, description)
resourceLeave(lrid)
resourceChange(lrid)
componentChange(lcid, description)
send(lbid, Object)
```

The information generated by resource join can trigger resourceReport(s) or be found through discoverResource calls. The leave and change calls generates resource and component reports, if there are subscribers. Send is used when component make calls on established bindings.

**Upcalls Initiated by Overlay Services**

```
result:allocate(lrid, description)
result:deallocate(lrid)
lcid:deploy(lrid, componentDescription)
bool:undeploy(lcid)
data:passivate(lcid)
lbid:bind(lcid, description)
bool:unbind(lbid)
lwid:watchComponent(lcid, eventDescriptions)
state:pollComponentState(lcid)
deliver(lbid, Object)
```

The allocate operation might consume the entire resource or just a part, in which case a new `rid` is given for the allocated part, while the old `rid` refers to the free remainder. The deallocate operation might return an instruction to merge two chunks of a previously split resource, or they might remain split.

# 6. Use Case

We demonstrate the use of the management overlay services with an application that consists of a single master component and multiple worker components. The master divides a computational task into independent subtasks, delivers each subtask to a random worker for processing taking advatage of the 'any' type of bindings, and collects and collates the results.

## 6.1 ADL Specification

```
definition MasterWorkerApplication
component Master
  content = MasterImpl;
  resourceSpecs requirements = "OS=Linux and MemorySize>4GB", preferences = "MemorySize";
  componentAttributes stateful,serializeable;
component Workers
  content = WorkerImpl;
  cardinality = 3;
  resourceSpecs requirements = "CPUSpeed>3GHz", preferences = "CPUSpeed";
  componentAttributes stateless;
binding B1
  client = Master.OutputInterface; server = Worker.InputInterface; type = any
```

A tool maps the ADL description to manager assembly code. This code contains invocations to the overlay services via the management API, as shown

in Section 6.2. The binding element of type “any” causes the invocations to be delivered to a single, random group member.

The management code for self-configuration and self-healing does not change the application’s architecture, and can therefore be generated from the same ADL specification automatically (see for example Section 6.3). For the master component, the ’componentAttributes’ field states that the component is stateful and therefore its state must be moved when the component is relocated, and that it is serializable meaning that the state can be actually saved into a data structure, as needed by e.g. the checkpointing code.

Self-tuning involves adjusting the number of workers when their load changes. The application ADL contains the policy ’self-tuning-workers’ describing this behaviour. The ’ComponentStateChange’ event specification causes setting up a watcher for a specific parameter of a source component using a given threshold value. The handler ’ManageGroupWithLimits’ changes the number of components in the ’Workers’ group such that the load moves into the region specified by “low” and “high” parameters.

```
policy self-tuning-workers
  event = ComponentStateChange(source=Workers, componentParam="Load", threshold="100")
  handler = ManageGroupWithLimits(target=Workers, low="1000", high="2000")
```

## 6.2 Initial Deployment

The following sections of management assembler code are produced automatically from the ADL descriptions. Waits are implicit; futures used as input parameters block calls until instantiated. Error-handling is omitted. The execution of the code is illustrated in Figure 1.

```
ridA:oneShotResourceDiscover(reqA, compare)
% + similarly for 3 B resources => ridB[1-3]
desA := specifications(preferenceA, ridA)
% specifications produces a description of how much of the
% resource is to be allocated
aridA:allocate(ridA, desA)
cidA: deploy(A, aridA)
% + similarly for 3 B components => cidB[1-3]
gid:group([cidB[1],cidB[2],cidB[3]])
bid:bind(cidA,gid, BDesA,BDesB, any)
% ’any’ indicates a one-to-some binding
widAR:watchResource(aridA, [used->fail, used->leaving], any, any)
activateEventHandler(self-config-leave, widAR, [bid, cidA, aridA, gid])
widA:discoverResource(reqA, compare, tCompare, 0, ridA)
activateEventHandler(self-config-join, widA, [bid, cidA, aridA, gid]
% do periodic check-pointing to enable self-healing for A:
timeGenerate(checkPoint(cidA), timeInterval)
activateEventHandler(checkpointing, id)
activateEventHandler(self-healing, widAR, [...])
activateEventHandler(self-tuning, widB[X], [...])
activateEventHandler(self-protection, widA, [...])
```

## 6.3 Self-* Rules

The rules are active until stopped, so they may be fired many times. Parameter names are left out when they are understandable from the context.

### Self-Configuration

```
upon event discoveryReport(wid, rid, RDes) with <bid, cidA, aridA, gid, preferencesA> do
  % the system reports a better resource match for A
  boolF: unbind(bid)
  pcid: passivate(cidA)
  boolF:deallocate(aridA)
  aridA: allocate(rid, specification(prefA, RDes))
  cidA:deploy(pcid, aridA)
  bid:bind(cidA, gid)
upon event resourceReport(widA, from, to) with <bid, cidA, aridA, gid, ...> do
    if from==used && to==leaving then
      newRidA:oneShotResourceDiscover(reqA, compare)
      boolF:unbind(bid)
  pcid:passivate(cidA)
  boolF:deallocate(aridA)
  aridA:allocate(newRidA, specification(preferencesA, newRidA))
  cidA:deploy(pcid, aridA)
  bid:bind(cidA, gid)
  % + update the widA-discoverResource with newRidA
```

Note that the aridA and other attributes are reset at rule termination so the rule can be fired again.

### Self-healing

```
upon event resourceReport(widA, from, to) with <?> do
  if from==used && to==failed then
    newRidA:oneShotResourceDiscover(reqA, compare)
  aridA:allocate(newRidA, specification(prefA, desc))
  cidA:deploy(fcidA, aridA) % fcidA from periodic checkpoint
  bid:bind(cidA, gid)
  % + update the widA-discoverResource with newRidA
```

### Self-Tuning

```
upon event componentReport(widB[X], load, oldLoad, newLoad) with <gid ...> do
  if loadMeasure(load[1..NoB])> HighLimit then
    % load is high we need another B
    newRidB:oneShotResourceDiscover(reqB, compare)
    newAridB:allocate(newRidB, specifications(prefB, desc)
    newCidB:deploy(B, newAridB)
    addToGroup(newCidB, gid)
    noB:=noB+1
    update(load[])
  elseif loadMeasure(load[1..NoB])<LowLimit then
    if noB > 3 then
      deallocate(lowestLoad(widB[]))
      noB:=noB-1
      update(load[])
```

**Self-Protection** Niche could check for unauthorized access requests on ports the VO has asked the user to open. An unauthorized request could be attempts to communicate without using established shared keys, in which case Niche could generate warnings.

```
upon event resourceReport(wid,unauthorized) with <...> do
  <generate warnings>
```

## 7. Related Work

Niche builds on a state-of-the-art overlay, DKS. A good survey of overlays and comparisons with DKS can be found in [9].

Several P2P-based publish/subscribe systems have been developed, e.g. [5, 7, 12, 18, 6]. To our best knowledge, those systems do not provide a robust and scalable event-notification mechanism (with high delivery guarantees) that could tolerate churn in highly dynamic Grids.

There is a considerable industrial and academic interest in self-* systems. Some approaches, e.g. [14, 11], rely on P2P to support some of self-* aspects in Grid. Our work aims at utilizing P2P to provide support for all self-* aspects in Grid component-based applications.

The AutoMate project [15] proposes a programming framework to deal with challenges of dynamism, scale, heterogeneity and uncertainty in Grid environments [13]. The Accord's approach for self-management is similar to our approach. Accord is based on the concept of an autonomic element that represents any (self-)manageable entity, e.g. resource, component, or object. The autonomic management in the Accord framework is guided by rules of two types: (i) behaviour rules, which control functional behaviours of autonomic elements and applications; and (ii) interaction rules, which control the interactions between elements and their environments and coordinate an autonomic application. The Accord framework provides support for run-time composition of elements and run-time injection of rules. In Accord, dynamic composition of autonomic elements is performed by a multi-agent infrastructure of the peer element managers and a composition manager. In [13], authors illustrate how the Accord framework can be used to realize each of the four aspects of self-management: self-configuration, self-optimization, self-healing and self-protection.

Our overlay services and approach to ADLs complement the AutoMate's architectural stack: AutoMate's self-* rules can be implemented using our work. Our work, however, does not stipulate a specific programming model and can be used to provide self-* management support for legacy applications.

The benefits of overlay-based middleware to support complex, heterogeneous, self-configuring Grid applications has been recognized in GridKit [11]. GridKit provides resource discovery and management services, and interaction services. However, GridKit does not provide specific services for component management, and did not address all of the self-* issues.

## 8. Discussion & Conclusions

In our architecture the overlay services form the middle layer of an infrastructure for the management of dynamic VOs. In particular overlay services are responsible for all sensing and actuation within the VO. They provide both push and pull provisions for sensing, including VO-wide discovery and sensor (de)installation on specific resources and components.

As the overlay services are based on a DHT, we have been leveraging results in the area been able to provide services that are both self-managing internally and robust. The fact that the overlay services are self-managing simplifies making the VO management itself. Within the limits of network connectivity a sensing event that takes place is always delivered to the management logic that has subscribed to it. Within the limits of network connectivity an actuation command that takes place is always delivered.

Dynamic VOs are characterized by high rates of churn and evolution so that there is real risk of overwhelming the management with a flood of status information and the need to take corrective actions. This might exceed bandwidth/storage/computation limitations and complicate dealing with management node failure. This is ameliorated by the following properties of the overlay services:

**Multiple management nodes**: The overlay services are potentially available to all nodes in the VO. Clearly there is a concurrency control issue when multiple management nodes act/sense on the same system elements; this is up to the management nodes to deal with.

**Abstraction**: Abstraction is the key to reducing the amount of sensing events and actuation commands that need to be handled by the management nodes. Churn may be hidden from the management. Examples are one free resource replaced by another, or when a component has been moved from one node to another. Via threshold functions we can ensure that only significant changes are reported by sensors. They may be configured to detect aggregate properties/events rather than individual properties/event. Examples are a watcher waiting to detect 5 free Linux resources to become available, and actuators acting on groups.

It needs to to be understood that the abstractions here are handled inside the overlay. An abstraction may either be an integral part of an overlay service or something that is executed exclusively on the management node(s). These two different designs can be made functionally equivalent. In both cases programming management may be simplified which is important for dynamic VOs as there are so many more basic events, such as resources joining and leaving, taking place than in static ones.

However, two designs are very different non-functionally. Abstractions incorporated in the overlay services make for fewer messages arriving at the management node, fewer messages overall in the system, and lower latencies. Our design criterion for incorporating abstractions into overlay services is that there is something to be gained non-functionally.

**Intelligent and dynamic delegation of management logic** is natural for hierarchical component models (e.g. deploying the self-management function of a subsystem of components on resource(s) near the components themselves). Also tasks of self-management may be split in aspect-oriented manner and

decentralized (e.g. self-healing in one place, self-tuning in another). Dynamic delegation means that the decision to delegate some management logic can be made at run time. One use case would be when the management node becomes heavily loaded that some management functionality can be delegated (e.g. the management node sends all logic associated with tuning to some other node, but keeps healing and configuration functionality in place).

Moreover, delegation can be made intelligent, in the sense that the delegated management logic may be placed optimally. In particular a management rule that is triggered by sensor events may be placed close to (or at) where the sensing events originate. Alternatively the management rule may be placed close to where actuation commands are to be sent to. There are, of course, cases where determining optimality is complex, depending on latencies between nodes and probabilities of various kinds of sensing events. At the same time there are rules where optimality determination is very easy. An example of the latter is a management rule that triggers on a resource leaving the system (e.g. member going off-line) which is best executed at the node containing the resource. The design is open-ended with respect to the intelligence of delegation.

This is work in progress, to date a number of overlay service functions have been designed, and some of these have been implemented.

**Acknowledgments** This research is supported by the European project Grid4All and the CoreGrid Network of Excellence.

## References

[1] Distributed k-ary system (dks). http://dks.sics.se/.

[2] L. Onana Alima, A. Ghodsi, P. Brand, and S. Haridi. Multicast in DKS(N,k,f) overlay networks. In *Proc. of the $7^{th}$ Int. Conf. on Principles of Dist. Systems*. Springer, 2003.

[3] L.O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS(N,k,f): A family of low communication, scalable and fault-tolerant infrastructures for P2P applications. In *$3^{rd}$ IEEE Int. Symp. on Cluster Computing and the Grid*, pages 344–350. IEEE, 2003.

[4] Bouchenak *et al.* Architecture-based autonomous repair management: An application to J2EE clusters. In *Proceedings of the 24th IEEE Symp. on Reliable Distributed Systems*. IEEE, 2005.

[5] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(8):1489–1499, 2002.

[6] R. Chand and P. Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In *Proceedings of EuroPAR 2005*, volume LNCS 3648, pages 1194–1204. Springer, 2005.

[7] P.-A. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl. Publish/subscribe for RDF-based P2P networks. In *Proceedings of the $1^{st}$ European Semantic Web Symposium*, volume LNCS 3053, pages 182–197. Springer, 2004.

[8] S. El-Ansary, L. Onana Alima, P. Brand, and S. Haridi. Efficient broadcast in structured P2P networks. In *Proc. 2nd Int. Workshop On Peer-To-Peer Systems*. Springer, 2003.

[9] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD thesis, Royal Institute of Technology (KTH), 2006.

[10] A. Ghodsi, L. Onana Alima, and S. Haridi. Symmetric replication for structured peer-to-peer systems. In *Proceedings of The $3^{rd}$ Int. Workshop on Databases, Information Systems and P2P Computing*, Trondheim, Norway, 2005.

[11] Grace *et al.* GRIDKIT: Pluggable overlay networks for grid computing. In *Proc. Distributed Objects and Applications (DOA'04)*, volume LNCS 3291. Springer, 2004.

[12] A. Gupta, O. Sahin, D. Agrawal, and A. El Abbadi. Meghdoot: content-based publish/subscribe over P2P networks. In *Proceedings of the $5^{th}$ ACM/IFIP/USENIX Int. Conf. on Middleware*, pages 254–273. Springer, 2004.

[13] H. Liu and M. Parashar. Accord: A programming framework for autonomic applications. *IEEE Transactions on Systems, Man and Cybernetics*, 36(3):341–352, 2006.

[14] Mayer *et al.* ICENI: An integrated Grid middleware to support E-Science. In *Proceedings of the Workshop on Component Models and Systems for Grid Applications*, CoreGRID series, pages 109–124. Springer, 2005.

[15] M. Parashar, Z. Li, H. Liu, V. Matossian, and C. Schmidt. Enabling autonomic grid applications: Requirements, models and infrastructure. In *Proceedings of the Conf. on Self-Star Properties in Complex Information Systems*, volume LNCS 3460. Springer, 2005.

[16] F. Schintke, T. Schütt, and A. Reinefeld. A framework for self-optimizing Grids using P2P components. In *$14^{th}$ Int. Workshop on Database and Expert Systems Applications*, pages 689–693. IEEE, September 2003.

[17] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, CA, August 2001.

[18] S. Voulgaris, E. Rivière, A.-M. Kermarrec, and M. van Steen. Sub-2-Sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. Technical Report RR-5772, INRIA, December 2005.